Intel® Inspector XE
2013

Memory Checker
Thread Checker
Static Analysis
Pointer Checker
Dynamic Analysis
Memory Errors
- Invalid Accesses
- Memory Leaks
- Uninit. Memory Accesses

Threading Errors
- Races
- Deadlocks
- Cross Stack References

Static Analysis
Code & Security Errors
- Buffer over/under flows
- Incorrect pointer usage
- Over 250 error types...

Pointer Checker
Pointer Errors
- Out of bounds accesses
- Dangling pointers

Deliver More Reliable Applications
Intel® Inspector XE and Intel® Parallel Studio XE family of suites

Find errors earlier with less effort

Static Analysis & Pointer Checker are only available in the Parallel Studio XE family of suites. Not sold separately.
Dynamic Analysis Finds Memory & Threading Errors
Intel® Inspector XE 2013

Find and eliminate errors
• Memory leaks, invalid access...
• Races & deadlocks
• C, C++, C#, F# and Fortran (or any mix)

Simple, Reliable, Accurate
• No special recompiles
  Use any build, any compiler
• Analyzes dynamically generated or linked code
• Inspects third party libraries where source is unavailable
• Productive user interface
• Command line for automated regression analysis

Easy to fit into your existing process

Clicking an error instantly displays source code snippets and the call stack
**New for 2013!**

Intel® Inspector XE 2013 Dynamic Memory & Thread Analysis

### Heap Growth Analysis

Diagnose heap growth. Get a list of memory allocations not freed in an interval set with the GUI or an API.

### Debugger Breakpoints

Diagnose the problem. Break into the debugger just before the error occurs. Examine the variables and threads.

### Improved Error Suppression

More precise and team shareable. Choose which stack frame to suppress. Eliminate the false, not the real errors.

### Pause/Resume Collection

Speed-up analysis by limiting its scope. Turn on analysis only during the execution of the suspected problem.

Find and diagnose errors with less effort.
## Pointer Checker and Memory Checker

Intel Parallel Studio XE family of suites

<table>
<thead>
<tr>
<th>Pointer Checker</th>
<th>Memory Checker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recompile with Intel® Compiler</td>
<td>Use any build, any compiler</td>
</tr>
<tr>
<td>Lower overhead</td>
<td>Higher overhead</td>
</tr>
<tr>
<td>Only finds pointer errors</td>
<td>Finds multiple error types</td>
</tr>
<tr>
<td>One error at a time</td>
<td>GUI sorts multiple errors</td>
</tr>
<tr>
<td>Traceback: Source file + Line #</td>
<td>Traceback: Shows source code</td>
</tr>
<tr>
<td>Triggers debugger breakpoint</td>
<td>Triggers debugger breakpoint</td>
</tr>
</tbody>
</table>

Two great ways to create more reliable software
Static Analysis Finds Coding and Security Errors

Intel® Parallel Studio XE 2013 Family of Suites

Find over 250 error types
- Incorrect directives, memory leaks, pointer and array errors, buffer overflows, uninitialized variables...

Easier to use
- Choose your priority:
  - Minimize false errors
  - Maximize error detection
- Hierarchical navigation of results
- Share comments with the team

Increased Accuracy & Speed
- Detect errors without all source files
- Better scaling with large code bases

Code Complexity Metrics
- Find code likely to be less reliable

Clicking an error instantly displays source code snippets and traceback. Available for C, C++ and Fortran.

Find Errors and Harden your Security

Static Analysis is only available in the Parallel Studio XE family of suites. It is not sold separately.
Dynamic Analysis Complements Static Analysis
In Intel® Parallel Studio XE family suites

<table>
<thead>
<tr>
<th>Dynamic Analysis</th>
<th>Static Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use any build, any compiler</td>
<td>Rebuild with Intel® Compiler (Keep your existing compiler for code generation.)</td>
</tr>
<tr>
<td>Fewer false errors. Only active code paths are analyzed.</td>
<td>Comprehensive, but more false errors. Not limited by test cases.</td>
</tr>
<tr>
<td>Analyze 3rd party code</td>
<td>n/a – Source required</td>
</tr>
<tr>
<td>Can trigger debugger breakpoint</td>
<td>n/a – No diagnostic capability</td>
</tr>
<tr>
<td>Slow (1x – 20x - 100x workload)</td>
<td>Fast (no workload, “slow” build)</td>
</tr>
<tr>
<td>Memory &amp; Threading Errors</td>
<td>Memory, Code &amp; Security Errors</td>
</tr>
</tbody>
</table>

Two great ways to create more reliable software
User Interface
Intel® Inspector XE

Select a problem set

Code snippets displayed for selected problem

Problem States:
New, Not Fixed, Fixed, Confirmed, Not a problem, Regression

Timeline shows when error occurred

Filters let you focus on a module, or error type, or…

Copyright © 2012, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners.
Double Click for Source & Call Stack
Intel® Inspector XE

Source code locations displayed for selected problem
Problem State Lifecycle
Makes problems easier to manage

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>Detected by this run</td>
</tr>
<tr>
<td>Not Fixed</td>
<td>Previously seen error detected by this run</td>
</tr>
<tr>
<td>Not a Problem</td>
<td>Set by user <em>(tool will not change)</em></td>
</tr>
<tr>
<td>Confirmed</td>
<td>Set by user <em>(tool will not change)</em></td>
</tr>
<tr>
<td>Fixed</td>
<td>Set by user <em>(tool will change)</em></td>
</tr>
<tr>
<td>Regression</td>
<td>Error detected with previous state of “Fixed”</td>
</tr>
</tbody>
</table>
Filtering - Focus on what’s important
Example: See only the errors in one source file

Before – All Errors

After – Only errors from one source file

(1) Filter – Show only one source file

(2) Error count drops

Tip: Set the “Investigated” filter to “Not investigated” while investigating problems. This removes from view the problems you are done with, leaving only the ones left to investigate.

Static Analysis shown, but filters work the same way for dynamic memory & threading analysis.
Command Line Interface

Automate analysis

inspxe-cl is the command line:

- **Windows:** C:\Program Files\Intel\Inspector XE \bin[32|64]\inspxe-cl.exe
- **Linux:** /opt/intel/inspector_xe/bin[32|64]/inspxe-cl

Help:

inspxe-cl -help

Set up command line with GUI

Command examples:

1. inspxe-cl -collect-list
2. inspxe-cl -collect ti2 -- MyApp.exe
3. inspxe-cl -report problems

Great for regression analysis – send results file to developer
Command line results can also be opened in the GUI
## Intel® Parallel Studio XE Suites
Leading development suite for application performance

<table>
<thead>
<tr>
<th></th>
<th>Intel® Cluster Studio XE</th>
<th>Intel® Parallel Studio XE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Analysis</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>● Intel® VTune™ Amplifier XE - Performance Profiler</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>● Intel® Inspector XE - Memory &amp; Thread Analyzer</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>● Static Analysis &amp; Pointer Checker - Find Coding &amp; Security Errors</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>● Intel® Advisor XE - Threading Assistant</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>● Intel® Trace Analyzer &amp; Collector - MPI Optimizing Tool</td>
</tr>
<tr>
<td><strong>Compilers &amp; Libraries</strong></td>
<td>●</td>
<td>● Intel® Compiler - Optimizing Compiler for C, C++ and Fortran</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>● Intel® Integrated Performance Primitives† - Media and Data Optimizations</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>● Intel® Threading Building Blocks† - Parallelize Applications for Performance</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>● Intel® Math Kernel Library - High Performance Math</td>
</tr>
<tr>
<td></td>
<td>●</td>
<td>● Intel® MPI Library - Flexible, Efficient and Scalable Messaging</td>
</tr>
</tbody>
</table>

† Available for C, C++ only

C, C++ only and Fortran only versions of Parallel Studio XE are also available.
Additional Material

Intel® Inspector XE

Product page for Intel Inspector XE and Static Analysis

Short demo & “how to” movies:
• Intel Inspector XE memory and thread checking
• Static Analysis correctness and security checking
• Cheat sheet on how to set up static analysis:  C, C++ and Fortran

Evaluation Guides – complete list
• Eliminate Memory Errors
• Resolve Resource Leaks
• Static Analysis for C, C++ and Fortran

Support - Search Support Articles

More products:  Intel Software Development Products
• Intel VTune Amplifier XE - performance and thread profiler
• Intel Advisor XE – threading assistant
Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY
ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS
DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR
IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES
RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY
PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on
Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using
specific computer systems, components, software, operations and functions. Any change to any of
those factors may cause the results to vary. You should consult other information and performance
tests to assist you in fully evaluating your contemplated purchases, including the performance of that
product when combined with other products.

Copyright © , Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, VTune, and Cilk
are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that
are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and
other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on
microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for
use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel
microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding
the specific instruction sets covered by this notice.

Notice revision #20110804
Backup
Dynamic Analysis Finds Hidden Errors Early  
Intel® Inspector XE 2013

Cross-thread Stack Access  
Occurs when a thread accesses a different thread’s stack.

Data Race  
Occurs when multiple threads access the same memory location without proper synchronization and at least one access is a write.

Deadlock  
Occurs when two or more threads are waiting for each other to release resources (such as mutexes, critical sections, and thread handles) while holding resources the other threads are trying to acquire. If none of the threads release their resources, then none of the threads can proceed.

GDI Resource Leak  
Occurs when a GDI object is created but never deleted.

Incorrect memcpy Call  
Occurs when an application calls the memcpy function with two pointers that overlap within the range to be copied. This condition is only checked on Linux® systems. On Windows® systems, this function is safe for overlapping memory.

Invalid Deallocation  
Occurs when an application calls a deallocation function with an address that does not correspond to dynamically allocated memory.

Invalid Memory Access  
Occurs when a read or write instruction references memory that is logically or physically invalid.

Invalid Partial Memory Access  
Occurs when a read or write instruction references a block (2-bytes or more) of memory where part of the block is logically invalid.

Kernel Resource Leak  
Occurs when a kernel object handle is created but never closed.

Lock Hierarchy Violation  
Occurs when the acquisition order of multiple synchronization objects (such as mutexes, critical sections, and thread handles) in one thread differs from the acquisition order in another thread, and these synchronization objects are owned by the acquiring thread and must be released by the same thread.

Memory Growth  
Occurs when a block of memory is allocated but not deallocated within a specific time segment during application execution.

Memory Leak  
Occurs when a block of memory is allocated and never released.

Mismatched Allocation/Deallocation  
Occurs when a deallocation is attempted with a function that is not the logical reflection of the allocator used.

Missing Allocation  
Occurs when an invalid pointer is passed to a deallocation function. The invalid address may point to a previously released heap block.

Thread Start Information  
Occurs when the Intel Inspector XE detects the creation of a thread. This problem is really informational feedback useful for confirming the number and location of threads created during application execution and data collection.

Unhandled Application Exception  
Occurs when the application undergoing analysis crashes because of an unhandled exception thrown by the application.

Uninitialized Memory Access  
Occurs when a read of an uninitialized memory location is reported.

Uninitialized Partial Memory Access  
Occurs when a read instruction references a block (2-bytes or more) of memory where part of the block is uninitialized.

For details, see our online documentation.
Static Analysis Finds Over 250 Kinds of Errors
Intel® Parallel Studio XE 2013 family of suites

Here are some examples...

- ALLOCATABLE array referenced before allocation
- Argument corresponding to * for width or precision value should be type int
- Argument count mismatch
  - Argument count mismatch at call to intrinsic function
  - Argument is not a pointer
  - Argument type mismatch at call to intrinsic function
  - Array parameter element size mismatch
- Array parameter rank mismatch
  - Array parameter shape mismatch
  - Attempt to violate exception specification
  - Bad format flags
  - Base class has non-virtual destructor
  - Base class lacks destructor
  - Big parameter passed by value
  - Bounds violation
- Buffer overflow through pointer
  - C library routine violates C++ object semantics
  - Chunk_size in OpenMP* SCHEDULE clause has side-effects
  - Chunk_size in OpenMP* SCHEDULE clause not loop-invariant
  - Class has virtual member functions but no derived classes
  - COMMON block is partly OpenMP* THREADPRIVATE
  - Conditional OpenMP* BARRIER
  - Data race
  - Data race from cilk_for
- Data race from cilk_spawn
  - Destructor contains non-empty exception specification
  - Divide by zero
  - Double free
  - Duplicate subroutine definition
  - Exception thrown from destructor
  - File closed twice
  - Format to argument count mismatch
  - Format to argument type mismatch
- FORTRAN IN argument modified
- Function illegally exits OpenMP* construct
  - Function result not set
  - Function return value discarded
- Function use does not match its definition
  - Gets function is unsafe
  - Global object constructor can throw exception
  - Global object destructor can throw exception
  - Global redefinition of new or delete
  - Global/static variable relies on default initialization
  - Illegal parameter value
  - Implicit function declaration
  - Implicit type conversion causes object slicing
- Improper nesting of OpenMP* constructs
  - Improper nesting of OpenMP* CRITICAL directives
  - Improper use of intrinsic function
  - Improper use of OpenMP* PRIVATE variable
  - Improper use of OpenMP* REDUCTION variable
  - Improper use of OpenMP* THREADPRIVATE array
  - Improper use of OpenMP* THREADPRIVATE variable
  - Inconsistent array declaration (element count mismatch)
  - Inconsistent array declaration (element size mismatch)
  - Inconsistent array declaration (element type mismatch)
  - Inconsistent array declaration (size mismatch)
  - Inconsistent enumeration declaration (enum value mismatch)
  - Inconsistent enumeration declaration (member count mismatch)
  - Inconsistent enumeration declaration (name mismatch)
  - Inconsistent enumeration declaration (tag mismatch)
  - Inconsistent enumeration declaration (type mismatch)
- Inconsistent pointer declaration (size mismatch)
  - Inconsistent pointer declaration (target size mismatch)
  - Inconsistent pointer declaration (type mismatch)
  - Inconsistent string declaration
  - Inconsistent structure declaration (field offset mismatch)
  - Inconsistent structure/union declaration (field count mismatch)
  - Inconsistent structure/union declaration (field name mismatch)
  - Inconsistent structure/union declaration (field size mismatch)
- Inconsistent structure/union declaration (field type mismatch)
  - Inconsistent structure/union declaration (size mismatch)
  - Inconsistent structure/union declaration (tag mismatch)
  - Inconsistent structure/union declaration (type mismatch)

For a more complete list, see our online documentation.